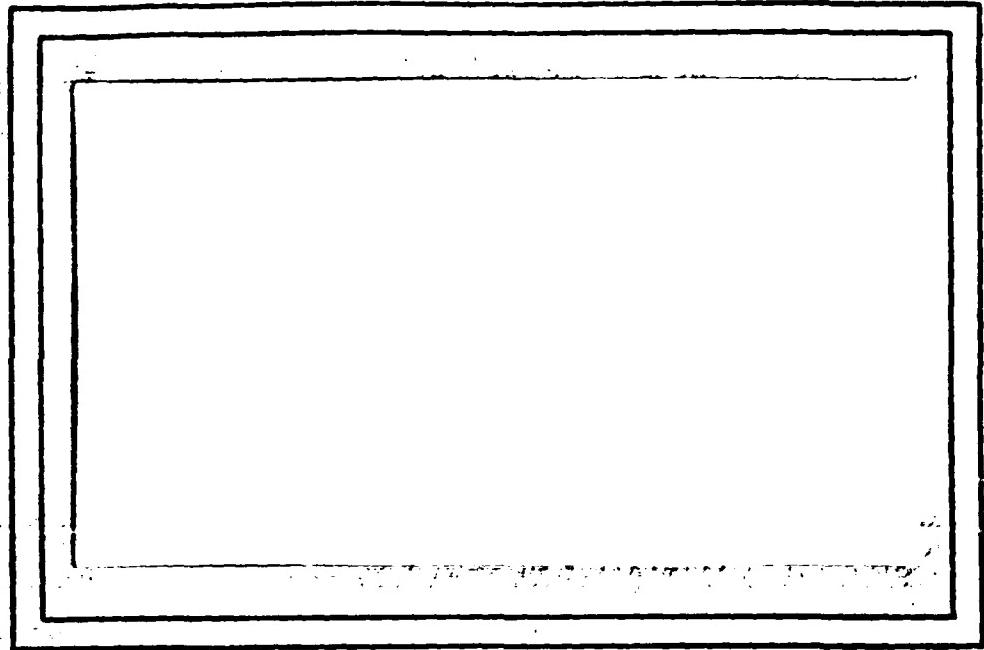


AD-A198 863

DTIC FILE COPY

(4)



COMPUTER SCIENCE
TECHNICAL REPORT SERIES



DTIC
ELECTED

AUG 10 1988

S H D

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND

20742

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

88 8 08 109

(4)

UMIACS-TR-88-32
CS-TR-2018

May, 1988

**Allocation of Real-Time Computations
under Fault Tolerance Constraints†**

Shem-Tov Levi and Daniel Mossé

Systems Design and Analysis Group
Computer Science Department

Ashok K. Agrawala

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

DTIC

ELECTRONIC

AUG 1 0 1988

ABSTRACT

Allocation of resources in "next-generation" real-time operating systems requires some important features in addition to those demonstrated by current systems, resulting in an increased complexity of each system. The allocation is closely related to the scheduling, and the two are based on *time* considerations, rather than on a static priority scheme. The allocation is fault tolerance motivated, to cope with the application's reliability goals. Distributed system issues and adaptive behavior requirements increase the complexity and significance of the allocation approach.

The allocation scheme we propose here accomplishes the hard real-time goal of guaranteeing a deadline satisfaction in case the job is accepted. In addition, this allocation scheme supports fault tolerance objectives in both damage containment and resiliency requirements. It does this in cooperation with a schedulability verification mechanism, and with an object architecture in which for each object there exists a *calendar* that maintains the time of its execution. A nice feature of this scheme is the way in which it can be used for reallocation while increasing the resiliency.

† This work is supported in part by contract N00014-87-K-0241 for the Office of Naval Research and by contract DSAG60-87-C-0066 from the US Army Strategic Defense Command to the Department of Computer Science, University of Maryland. The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Navy or the Army position, policy, or decision, unless so designated by other official documentation.

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

Contents

1	Introduction	4
1.1	Objects Architecture	4
1.2	Guarantees in Hard Real-Time Systems	5
1.3	Scheduling Feasibility Verification	7
2	Problem Definition and Formulation	8
2.1	Review of Some Allocation Approaches	8
2.1.1	Allocation with IPC Minimisation	8
2.1.2	Allocation with Bottleneck Processor Load Minimization	9
2.1.3	NEXT-FIT-M Partitioning for Rate Monotonic Schedulers	10
2.1.4	Allocation with Load Balance Optimality Constraint	11
2.1.5	Heuristic Approaches	13
2.2	Model Description	14
2.3	Conditions and Formulation	16
3	Allocation Algorithm	19
3.1	Message Types Used	19
3.2	Principles of Algorithm for Initiator	20
3.3	Principles of Algorithm for Allocator	21
3.4	Local and External Variables	23
3.5	The Allocation Algorithm	23
4	Properties	28
4.1	Algorithm Termination	28
4.2	Allocatability Correctness	29
4.3	Achievement of Fault Tolerance Objectives	29
4.4	Complexity	30
4.5	Mutual Exclusion	31
4.6	Deadlock	31
5	Reallocation Algorithm	31
5.1	Rationale	31
5.2	Algorithm for Detection Unit D_i	32
6	Concluding Remarks	32
A	Time Constraints and Auxiliary Variables in Object's Joint	34
B	Detailed Review of Allocation Algorithms	35 or
B.1	Bottleneck Load Minimisation Allocation Algorithm	35
B.2	Next-Fit-M Allocation Algorithm	36
B.3	Algorithm for Relocation upon Failure Detection	37
B.3.1	Relocation within The Detection Unit	37

B.3.2	Algorithm for Relocation to Another Detection Unit	38
B.4	Heuristic Scheduling Algorithm	39
C	Scheduling Functions Used by The Fault-Tolerant Allocation	40
C.1	Inserting Time Constraint into Object's Calendar	40
C.2	Removing Time Constraint from Object's Calendar	41
C.3	Loading and Unloading Time Constraint in Object's Calendar	42

1 Introduction

This paper examines the problem of allocating resources and computation services to support the execution of a distributed hard real-time computation. Allocation of resources in "next-generation" real-time operating systems requires some important features in addition to those demonstrated by current systems, resulting in an increased complexity of each system. The new systems must provide for general distribution issues, like deadlock prevention, along with supporting adaptive behavior requirements. In addition, there are requirements to support hard real-time goals and fault tolerance objectives. These goals and objectives must be guaranteed to be satisfied even under specified environment changes.

In real-time operating systems the resource allocation has to be related closely to the scheduling, and the two are based on time considerations. Scheduling is the mechanism through which the timing properties of an execution instance of a software module are finalized. The allocation must also be fault tolerance motivated, to cope with the reliability goals of an application.

The allocation scheme we propose here supports fault tolerance goals in both damage containment and resiliency requirements. In addition, this allocation scheme accomplishes a deadline satisfaction guarantee for all its accepted jobs. It does this in cooperation with a schedulability verification mechanism, and with an object architecture, in which for each object there exists a *calendar* that relates time to its execution.

The paper is organised as follows. In the remaining of this section we review briefly the object architecture and the schedulability verification ([1,7,8]) that support hard real-time environments. We also introduce some tools that help us deal with the allocation problem. In the following section we formulate the problem and the conditions for a solution. We then introduce an algorithm that implements the above solution, and finally we investigate some of its properties. The paper closes with some concluding remarks.

1.1 Objects Architecture

In [1,7] we have introduced an architecture for designing hard¹ real-time operating systems. This architecture is based on the use of highly encapsulated entities, called objects. An object is a distinct and selectively accessible software element that resides on one of the storage resources of the system. The objects architecture defines the objects as the elements that constitute the system. It also defines their classification, the relationships between them, the set of operations they are subjected to and execution parameters that permit scheduling them for execution and access.

In software engineering context, an object is viewed as an entity whose behavior is characterised by the operations it is subjected to and the operations it carries out on other objects. The external view of an object (these operations) is its specification, and the internal view of an object is its implementation. In our architecture, we have considered the use of object architecture in a system context, thereby expanding the above object definitions to describe elements and entities in a more general way. Yet, some of the properties that characterise objects in software development context are valid in the system architecture context as well. For example:

- An object has a state.
- There is a set of actions to which an object is subjected and a set of actions it requires from other objects.

¹Hard real-time systems are characterized by their property of having a nonrecoverable fault when a computation does not complete before its deadline.

- It is denoted by a name.
- It has restricted visibility of (as well as by) other objects.

We have shown that our object-oriented system design methodology provides means to construct systems with a high degree of deterministic and predictable timing properties [1,7]. This determinism, together with the required fault tolerance schemes, are major principles in our time-constraint oriented system. We have defined a classification of object types, the set of operations each of the object types is associated with, and their relationships. A conceptual model has been considered in our analysis of the applicability of objects architecture for a real-time, distributed, and fault tolerant operating system. Issues of creation, deletion and access for manipulation and state verification, have lead us to define the *joint* that consists of the following parts:

- A context independent pointer to the object's body, enabling the naming network to support a multi-user, selective sharing of the object.
- An owner/user justification structure.
- Resource (and/or server) requirements.
- A ticket check mechanism for the protection scheme.
- A time constraint for an executable object.
- A replica/alternative control mechanism for the fault tolerance scheme.

In our model, objects that relate to each other are connected via the owner/user justifications in the joint. These relationships are in accordance with the visibility restrictions and the set of operations to whom the justicand is subjected and the justifier operates on. Operations in this set can change an object's state, evaluate current state of an object, and allow visiting parts of an object. These operations can be carried out on object bodies as well as on object joints. We can model a system as a graph whose nodes are objects and whose arcs are directed from justifier to justicand representing the owner/user relationship. Relationships between objects necessitate the grouping of objects of the same type into a meta-object, to which the rest of the objects may refer to as a whole entity.

Scheduling executable objects and context initialisation are divided in our model into an on-line part and an off-line part. The context initialiser consists of an off-line *allocator/binder* that manages the acceptance of jobs (requests to execute objects) and allocates the resources before loading, and an on-line *loader* that activates schedulable objects that are invoked. The scheduling policy is managed by the *off-line scheduler*, which is responsible for recognising the availability of resources, and provides scheduling feasibility verification testing and reservation facilities. The *on-line scheduler* carries out locally the policy, the dispatching and the preemption of loaded executable objects (*processes*) according to their time constraints.

The way in which the above allocator works is the major concern of this paper.

1.2 Guarantees in Hard Real-Time Systems

When a request for a specific object invocation arrives at a hard real-time reactive operating system, the operating system has to allocate (if feasible) all the resources required such that it is guaranteed that the

object's time constraint is met. Informally speaking, a time constraint is a requirement to start executing a particular executable object, after a condition is satisfied, and complete the execution before its deadline. The execution time of the object is assumed to be given, and the constraint is extended to a periodic execution of the object. Based on previous works that define hard real-time systems (e.g., [13,14]), we have defined a time constraint formally in [1,7] as the quintuple

$$< Id, Taft(condition_1), c_{Id}, f_{Id}, Tbef(condition_2) >$$

where:

Id is the name of the executable object (process) in the proper context,

$Taft(condition_1)$ states after what event should execution begin,

c_{Id} is the computation time of object Id ,

f_{Id} is the frequency with which the computation should be carried out,

$Tbef(condition_2)$ states the deadline which should be met.

The time interval defined by $Taft(condition_1)$ and $Tbef(condition_2)$ is the occurrence interval, which delimits the time domain in which the executable object is allowed to execute. In an interval-based notation, as we have used in [8], the occurrence interval and the computation interval relate to each other such that the above quintuple is supported. The occurrence interval is a convex (contiguous) interval, and the computation interval can be non-convex (since it may contain gaps). Let the j 'th occurrence of time constraint i be denoted as the convex interval $TC_i^{(j)}$, and let $P_i^{(j)}$ be a non-convex interval that represents the union of all possible execution traces of this computation. Then,

- $Taft(condition_1) \longrightarrow \text{begin}_{\min}(TC_i^{(j)})$.
- $Tbef(condition_2) \longrightarrow \text{end}_{\max}(TC_i^{(j)})$.
- f_i periodicity $\longrightarrow \forall j > 1 : \text{end}_{\max}(TC_i^{(j)}) - \text{end}_{\max}(TC_i^{(j-1)}) = \frac{1}{f_i}$.
- c_i computation time $\longrightarrow \forall j \geq 1 : \|P_i^{(j)}\| = c_i$.

A real-time operating system must use the time constraint as the key to its decisions on execution initiation and resource scheduling ([13]). Before execution initiation, the allocation and context initialization are required to ensure schedulability of an accepted job. In other words, before loading an object, a positive feasibility result must show that there exists a schedule that includes the invoked object, according to its time constraint, with no conflict with the already accepted objects. All the required resources should be allocated and reserved for the object, assuring that it is going to meet its deadline. It is not only processors that are to be allocated for an object. An object may rely on other server objects in order to perform its functions. These other objects may or may not reside at the same site. Remote services necessitate the needs for agents and for communication, each of which has to be schedulable within the time constraints of the invoked object.

One must take into account the time constraints that are projected between different computation localities, such that each computation locality might have access to a different clock with a different accuracy

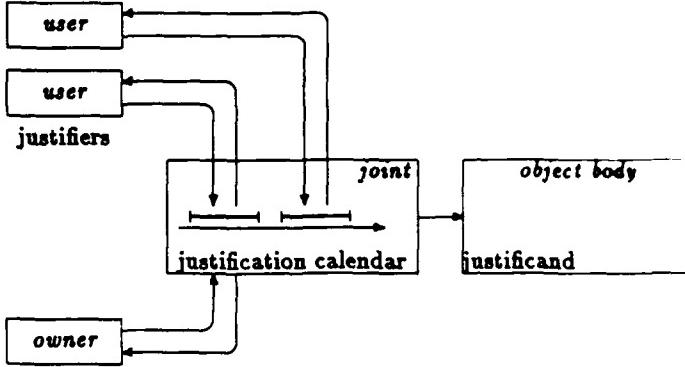


Figure 1: Object's Owner and User Temporal Justification

and correctness. This projection is discussed in details in [8]. A chosen (and loaded) allocation should avoid conflicts when users share a server object, with respect to violations of the user objects' time constraints. Each server object is then considered as a resource, and maintains its own schedule. When a server is allocated to a new user, the binder updates the justification links. The allocated server's future schedule is to be checked to show schedulability within the new user's time constraints, without violating the services this server has already guaranteed to serve.

1.3 Scheduling Feasibility Verification

In [8] we have introduced formulation and algorithms that verify the feasibility of scheduling an incoming execution request, while maintaining and scheduling the requests which have already been accepted before. These algorithms are based on the architecture described above, and the mechanisms that support the above algorithms.

Figure 1, describes a temporal justification scheme which is used in our architecture. The same server object is allocated to different users at different times, hence creating a user justification for this object at different time intervals. These intervals are in the future and according to them real-time scheduling decisions are to be taken. A very important issue arises in the above justification scheme. The time according to which the decisions are taken is a local and imprecise view of the global time. Distributed computations may have the same local view at different nodes at different "real" times. Therefore, future projection of time has to avoid ambiguities and conflicts that originate in differences between local views.

Upon arrival, the time constraint of each incoming request is properly projected, and tested for a possible insertion into the required object's calendar. The test for schedule feasibility depends on the scheduling policy employed. Conditions for schedule feasibility and algorithms for both preemptive and non-preemptive policies are introduced in [8]. In these algorithms, if there is a feasible schedule, then the time constraint of the incoming request is inserted in the calendar, reserving the computation interval for it in order to avoid ambiguity of answers to requests in contention. The space is reserved in a way that ensures that the required object will not be activated by the scheduler, unless an acknowledgement is sent by the initiator of this request. Furthermore, the reservation is kept for a limited time, and after this "timeout" elapses without

initiator acknowledgement, this request is removed from the calendar. On the other hand, if the test is negative, i.e. there is no feasible schedule that does not have a conflict with already guaranteed acceptances, then a negative answer should be given to the initiator of the incoming constraint. The initiator, in turn, should remove other requirements that have already been reserved at other calendars (if any), in case this negative answer prevents its execution.

The allocation model that is presented in this paper assumes the existence of the above mechanism. It is shown in [8] that this mechanism allows guaranteeing deadline satisfaction. In addition, it is shown later in this paper that this mechanism not only supports mutual exclusion from the contention point of view, but it also prevents deadlocks that might arise from some cyclic dependency in the global computation graph.

2 Problem Definition and Formulation

The problem of allocating the execution of computation elements to computation resources has been studied with respect to many dimensions of that problem. In most of the cases we have found in the literature, the goal of the allocation has been an optimisation of some metric of the execution performance, generally one stochastic parameter of the performance description. The model which has been mostly used in the above cases, reflected an allocation of processes to processors, while both the set of processes and the set of processors have been subjected to some inter-set relations and intra-set optimality constraints. In addition, in most of the above cases the nature of each of the processors was homogeneous, indivisible, and self contained.

We start this section with a review of some recent important works that have to do with allocation of real-time computation elements under high reliability requirements. We then introduce our computation model and our allocation goals, and define the requirements and the conditions for these goals to be met. A brief review of the graph properties we use later is given for the reader's convenience.

2.1 Review of Some Allocation Approaches

2.1.1 Allocation with IPC Minimization

A centrally controlled allocation scheme is described in [10,11], a scheme which has been used in the BMD project. There, a nominated computation node has the knowledge of the global status of the system, and each request for task execution passes this nominated node to be properly allocated to resources. The model uses only tasks and processors. The relations between tasks and processors are given in matrices, according to which the allocation is done.

- Task Preference Matrix P where $p[i, j] = 0$ means that task i is not allowed to execute on processor j .
- Task Exclusion Matrix X where $x[i, j] \neq 0$ means that task i and task j cannot be assigned to the same processor.
- Task Coupling Matrix C where $c[i, j]$ are the coupling factors that represent the amount of data transferred from task i to task j .
- Task Distance Matrix D where $d[i, j]$ represent the cost of transferring one data unit from task i to task j .

The allocation is considered as a search tree. In this tree, each vertex is a task to be allocated, and each arc that leaves the vertex is a possible allocation of a processor to that task. The search algorithm is based on the branch and bound method, and is constructed in setting and backtracking phases. The search goal is an allocation that minimizes the execution time of a "port-to-port" thread of executing tasks. The execution cost function consists of the following ([10]):

1. Execution time of the task on the processor, which depends on the task size and the processor MIPs rate:

$$E_i = \frac{\text{size(task } i\text{)}}{\text{processor MIPs rate}}$$

2. The network and operating system overhead (Ov), which is used for concurrency control, integrity checking, recovery check-point update, etc.
3. Inter-processor communication (IPC), which is higher if communicants reside on different processors.
4. Waiting time (WT) which is consumed when the task waits in the processor enablement queue. This figure depends highly on the sizes and number of tasks, the processor load, and the number of enablements. (Especially if large tasks are assigned to the same processor.)

The search algorithm ([11]) eliminates search in improper subtrees, while branching to a new subtree according to matrix P , matrix X , and the maximal capacity of each assigned processor. Preference is imposed with dominance relations, Ov , and WT . The IPC cost is computed for each subtree through matrices C and D , in $\sum c[i, j] \cdot d[i, j]$. The lowest cost solution is chosen out of the set of possible solutions.

The major disadvantages of this algorithm are its centralistic nature and the requirements for global knowledge. Furthermore, no time constraints are taken into account at a task level, and no fault tolerance goals are set. The above disadvantages indicate that this allocation scheme is not suitable for "next-generation" real-time operating systems.

2.1.2 Allocation with Bottleneck Processor Load Minimization

In [2], an objective function is suggested for the problem of allocation of tasks to processors, using an optimization constraint of minimizing the load on the bottleneck processor, i.e. the most heavily loaded processor. The algorithm is presented in section B.1.

The algorithm assumes that the load on a processor is a function of the inter-module communication (IMC), the accumulative execution time (AET) of the modules, and the precedence relations (PR) between executing modules. It defines the problem on a set of J modules, p_1, \dots, p_J , and a set S processors. The AET of module p_j during a particular time interval can be derived from the number of times p_j executes during this interval, and the average execution time of p_j over peak load periods. The AET s of the modules are assumed to be known, and are denoted as $\{T_j : 1 \leq j \leq J\}$. IMC in this approach incurs the inter-process communication cost (IPC) and the processing overhead. IPC can be significantly reduced if the allocation assigns pairs of heavily communicating processes to the same processor. The workload on a given processor (P_r), under a given assignment of the J modules to the S processors, is defined as

$$\mathcal{L}(P_r; X) = AET(P_r; X) + IMC(P_r; X)$$

where X is an assignment matrix $[x_{i,j}]$, for which $x_{i,j} = 1$ if p_i is assigned to processor j .

The assumptions taken in [2] on IPC , allow the selection of a model of communication cost as a sum of the cost of outgoing messages and the cost of incoming messages, whereas the costs of module enablements and control messages are ignored.

$$IMC_{i,j}(\mathcal{X}) = IPC(i, j; \mathcal{X}) + IPC(j, i; \mathcal{X}).$$

Given the average inter-module communication cost at peak load periods, $\{IMC_{i,j} : 1 \leq i, j \leq J\}$, which can be calculated from the volume of the communication between the modules, one can derive the IMC index

$$\gamma_{IMC}(i, j) = \frac{IMC_{i,j}}{AET}.$$

Thus, the load at r can be expressed as

$$\mathcal{L}(P_r; \mathcal{X}) = \sum_{j=1}^J x_{j,r} T_j + \sum_{s=1, s \neq r}^S IPC(r, s; \mathcal{X}) + \sum_{s=1, s \neq r}^S IPC(s, r; \mathcal{X}).$$

The bottleneck processor load is

$$bottleneck(\mathcal{X}) = \max_{1 \leq r \leq S} \{\mathcal{L}(P_r; \mathcal{X})\}$$

and minimizing this load is

$$\min_{\mathcal{X}} \{bottleneck(\mathcal{X})\}$$

or

$$\min_{\mathcal{X}} \left\{ \max_{1 \leq r \leq S} \{AET(P_r; \mathcal{X}) + IMC(P_r; \mathcal{X})\} \right\}.$$

Precedence relations (PR) affect the response time of the system, and this aspect is included in this algorithm. In [2], a model of wait-time behavior is constructed, based on the observed relation between size ratio of modules, $\rho_{i,j}$, and wait-time ratio, $R(\rho_{i,j})$. The algorithm then uses the PR index

$$\gamma_{PR}(i, j) = 1 - R(\rho_{i,j}).$$

The algorithm in section B.1 presents an iterative approach in which the workload $\mathcal{L}(P_{bottleneck}; \mathcal{X})$ is recorded for different tuning scale factors α and β . α represents the scale factor of combining γ_{IMC} with γ_{PR} , and β is a scale factor for the threshold of processor load on which combining is decided.

The $P-I-A$ algorithm disregards loads on processors due to other computations, rather than p_1, \dots, p_J , and therefore in order to allow independent computations it requires an extra knowledge. Being centralistic itself, implies that this global knowledge must be centralistic or static. Two major properties of hard real-time systems are not dealt here, since no time constraints are imposed on module execution, and no fault tolerance objectives are defined.

2.1.3 NEXT-FIT-M Partitioning for Rate Monotonic Schedulers

In [4], an on-line algorithm of $O(n)$ time complexity and $O(1)$ space complexity is introduced, to partition a set of tasks such that each partition will be scheduled later for execution at a distinct locality by a rate-monotonic priority scheduling algorithm. The allocation is centralistic, while the scheduling is distributed. A subgoal of the algorithm is to use as few processors as possible.

In this model, tasks have the following characteristics:

- Each task has a constant period.
- Each task has a deadline constraint, and no begin-time constraint is imposed.
- Tasks are independent of each other, without precedence constraints.
- All the tasks require the same computation time-interval.

The tasks are partitioned according to their *duty cycle*, which is the ratio between the identical computation interval and the task's period. They are then assigned to processors such that the processors will schedule them in a rate monotonic scheduling algorithm. Each of the rate monotonic algorithms is known to be bounded ([9]), and therefore the allocation maintains the load allocated to each processor such that it does not exceed that bound.

The allocation algorithm is described in section B.2. The tasks are divided into M classes, such that

- task $T_i \in \text{class_}k$ if $2^{\frac{1}{k+1}} < u_i \leq 2^{\frac{1}{k}}$ for $1 \leq k < M$.
- task $T_i \in \text{class_}M$ if $0 \leq u_i \leq 2^{\frac{1}{k}}$.

The algorithm assigns k class- k tasks to each class- k processor, keeping the utilization factor of the class- M processor less than $\ln(2)$.

The partitioning mechanism of this allocation algorithm is based on the use of local rate-monotonic priority schedulers, and it is therefore totally scheduler dependent. Even so, the model of the above scheduler is too simple to support "new-generation" real-time applications. The absence of begin-time constraints, the lack of support for a variety of computation requirements, the absence of important relations between tasks (precedence and others), and disregarding loads on processors due to other computations, are features that this approach fails to demonstrate. Furthermore, it fails to support any fault tolerance goals, and thus does not give a comprehensive solution. However, we find the relationship demonstrated in [4] between an allocator and local schedulers important and useful.

2.1.4 Allocation with Load Balance Optimality Constraint

In [6,5], allocation of processes to processors is examined with respect to a distributed load balancing optimality constraint, and groups of processes are relocated when one or more processes fail.

A set of processes $P_p = \{p_1, \dots, p_J\}$ are related to each other through a set of logical links \mathcal{L}_p , to form a graph

$$\mathcal{G}_p = (P_p, \mathcal{L}_p).$$

A set of processors $P_P = \{P_1, \dots, P_S\}$ are related to each other through a set of physical links \mathcal{L}_P , to form a graph

$$\mathcal{G}_P = (P_P, \mathcal{L}_P).$$

Each node in the above two graphs can be measured according to its incoming and outgoing links, applying some weights to the links to express communication costs. These measures can be used as similarity (clustering) measures, according to which each of the graphs is represented by a cluster tree, τ_p and τ_P , respectively. The allocation algorithm in [6] is mapping the nodes in τ_p to the nodes of τ_P . The motivation of this allocation is to assign heavily communicating processes to heavily connected processors.

In this approach, all the processes are assumed to be roughly equal in the load they impose when assigned to a processor, and this load is assumed to be a unit load. Each processor P_i is assumed to have a current assigned load denoted c_i . The current load is bounded by the processor capacity C_i , and is required to satisfy an optimality workload constraint

$$\left| \frac{c_i}{C_i} - \lambda \right| \leq \epsilon,$$

where λ is the optimal load and ϵ is a tolerance. The relation

$$m_i \leq c_i \leq M_i$$

is another way to express the optimality constraint, where

$$m_i = C_i \cdot (\lambda - \epsilon), \quad M_i = C_i \cdot (\lambda + \epsilon).$$

When a cluster of processors is observed, the sum of its processors' capacities expresses the cluster's capacity, and a sum of the currently assigned processor loads is the currently cluster load. A metric that represents the violation of optimality in cluster j can be expressed as

$$V_j = \left| \frac{c_j}{C_j} - \lambda \right| - \epsilon.$$

The ALLOCATE algorithm presented in [6] uses the violation values of the children of a node in processor cluster tree in order to select a candidate cluster of processors to which processes are to be assigned. The highest violation is selected first.

In order to support fault tolerance objectives, the occurrence of a fault must first be detected. A cluster of processors that monitor each others status and participate in the detection algorithm are called a *detection unit*. The set of S processor is therefore divided to detection units D_1, \dots, D_K , and each detection unit D_i is assigned with N_i processes.

Each detection unit is ordered, to have a *Leader*, second in command, etc. The assumption that no failure occurs while recovering from a previous failure, allows replacing a Leader that has failed using a simple protocol. Each Leader maintains some knowledge in order to answer questions of other Leaders that cannot relocate in their own detection unit. Each Leader maintains additional information for its relocation management, both for relocating locally (within the detection unit) and for relocating externally (moving processes to another detection unit).

When a processor P_j is detected to have failed in detection unit D_i , its capacity is removed from the total capacity of the detection unit. The Leader of D_i checks if the fault can be dealt with locally, by a local reconfiguration of the allocation within D_i . If this is the case, then

$$m(D_i) - m_j \leq N_i \leq M(D_i) - M_j$$

and the actions taken are described in section B.3.1. The benefits of a local relocation are the isolation of failures from other detection units and the minimisation of enforcement of departing from optimality.

However, if it cannot be treated locally, the Leader must generate a candidate set, of (v, k) pairs, to select both the node v to be migrated, and the destination detection unit D_k . For each node v to be migrated to detection unit D_k , three cost issues are raised:

1. *migration cost*, $M(v, k)$, which mainly consists of a fix overhead, and a cost which is proportional to the number of leaf nodes that are descendants of v ,
2. *affinity cost*, $A(v)$, which originates in the increased logical communication between D_i and the migration destination, and therefore depends on the links of the migrated node,
3. *utilization cost*, $B(v, k)$, which is a measure of the unbalanced load and the violation of the optimality measure in D_i and in the destination detection unit.

Combining the above three costs to a ranking measure, yields

$$R(v, k) = C_1 \cdot A(v) + C_2 \cdot B(v, k) + C_3 \cdot M(v, k).$$

An algorithm which is motivated by the above ranking is described in section B.3.2.

We find the relocation approach very appealing from fault tolerance point of view. However, this approach does not satisfy hard real-time system requirements, because it does not take into account the deadlines in its clustering measure (e.g. in ALLOCATE) and the impossibility to recover through a roll-back (e.g. in its migration solution) in many cases.

2.1.5 Heuristic Approaches

In general, the mapping of timing constraints plus the precedence relations onto resource allocation in a multi-processor environment is an NP-hard problem ([13,14]). This fact motivates the research for heuristics that provide sub-optimal solutions for the hard real-time allocation and scheduling problems. Some heuristic approaches taken in scheduling ([17,18]), suggest some interesting ideas with respect to the allocation scheme. An example of a heuristic scheduler, the one used in the Spring operating system, is given in section B.4.

In [17], at each level of the search tree, the scheduler updates a vector of the *Dynamic Resource Demand Ratio*

$$DRDR = (DRDR_1, \dots, DRDR_i, \dots, DRDR_r)$$

whose component " i " indicates the fraction of resource R_i to be used by the tasks not yet scheduled.

$$DRDR_i = \frac{\sum_T (c_T : T \text{ remains to be scheduled} \wedge T \text{ uses } R_i)}{\max_T (d_T : T \text{ remains to be scheduled} \wedge T \text{ uses } R_i) - EAT_i}$$

where EAT_i is the earliest available time of resource R_i , and d_T and c_T are the deadline of task T and its computation time respectively. One should notice that all the resources are reserved for the whole computation time. When a search decision is to be taken regarding the schedule feasibility, as in

if strongly-feasible(task_set, schedule) then ...

one should check also

$$\forall_{i=1, \dots, r} : DRDR_i \leq 1.$$

In [18], the scheduler allows preemption and thus each resource is allowed to be required in one of the following three modes:

- exclusive,

- shared,
- not needed.

At each level of the search tree, the scheduler updates another vector of the *Minimum Resource Demand Ratio*

$$MRDR = (MRDR_1, \dots, MRDR_r)$$

whose component "i" indicates the fraction of resource R_i to be used by the tasks not yet scheduled.

$$MRDR_i = \frac{\sum_{T \in R_i^{\text{not scheduled}}} (c'_T : T \text{ remains to be scheduled}) + \max_{T \in R_i^{\text{not exec}}} (c'_T : T \text{ remains to be scheduled})}{\max_T (d_T : T \text{ uses } R_i \wedge c'_T > 0)} - EAT_i$$

with the terms defined as above, except for c'_T which is the remaining execution time of task T .

2.2 Model Description

Our model of computation is a system constructed from objects and resources. The objects that participate in a computation are related to each other via semantic links that are pointed by the object joints. The temporal properties of each relation are expressed as either convex or non-convex time intervals in a calendar within the relevant joint. In that respect, resources also can be viewed as objects. However, we distinguish between the two for differences in fault tolerance properties that are related to monotonicity of faults. The distinction is also related to properties that concern damage containment in case of faults. The properties of the resources may allow us to model the system elements in terms of resource segments. For example, we may model one particular memory page as a resource if we can detect a failure at this level of resolution, and trigger an off-line recovery at the same level. On the other hand, if we cannot do the above, we may model the whole memory at a given locality as a resource, or even the whole locality (i.e. the processors, the memory, the devices, etc.) as a single resource.

Executing and "to-be-executed" objects are to be allocated as system resources, each having its own joint and calendar. These resources are physically linked according to their geographic and hardware constraints. However, in addition to resources, services provided by objects may need to be allocated to other objects. Each of these services may need other services and resources, and so on. We present this as a graph, where objects and resources are represented as nodes, and the relations are represented as directed arcs. Note that resources are always the leaf-nodes, since a resource is not expected to need services from other resources.

The distinction between transient and monotonic faults, as expressed in our object/resource model, allows the use of two possible recovery mechanisms. We denote the most common one as *temporal redundancy*, in which a "retry" effort is executed upon a fault detection. This mechanism is perfectly suited for faults whose existence may be a transient phenomenon. It also permits roll-back recovery. Real-time constraints may conflict with temporal redundancy, because the time needed for recovery may not exist. Furthermore, in case of a monotonic failure, retrying is ineffective. In such cases only *physical redundancy* can increase the system resiliency. Roll-forward recovery and the N-version programming are examples of such redundancy.

In Figure 2 we give an example of the two mechanisms. Objects a and b are allocated with temporal redundancy, while object c has a physical-redundancy in object d . In the model defined below, resources are to be subjected only to physical redundancy, while redundancy of objects is defined by the computation designer.

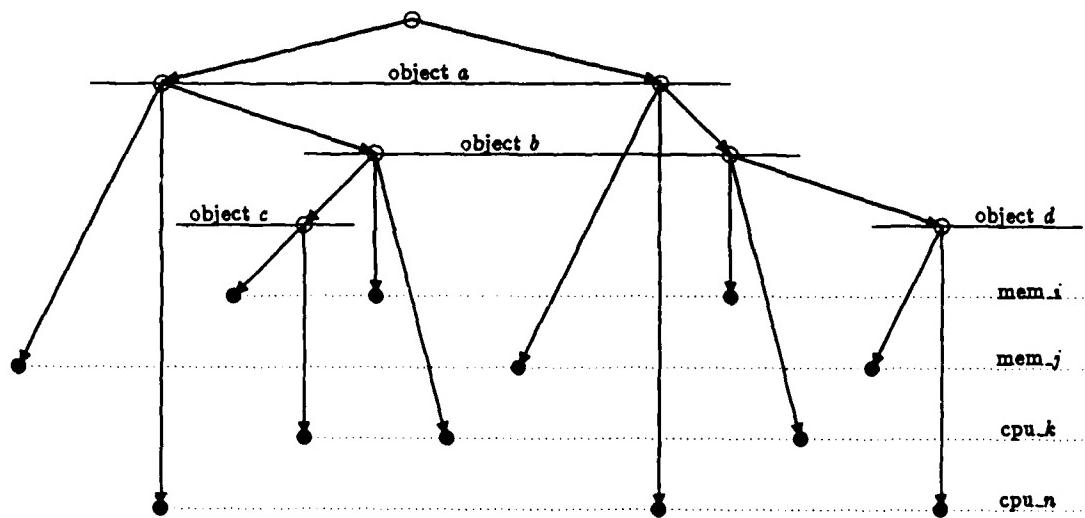


Figure 2: Temporal Redundancy (a,b) and Physical Redundancy (c/d)

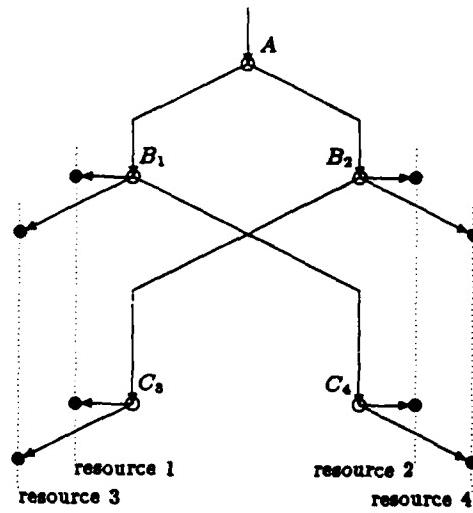


Figure 3: Wrong Use of Resources: 0-Resiliency

One major obstacle that the allocation and relocation mechanisms must overcome is shown in Figure 3. Although objects B_1 and B_2 are physically redundant, and so are objects C_3 and C_4 , the allocation in the figure results in a 0-resilient computation. Any failure of one of the four resources result in a computation fault, since both redundant threads depend on all four resources. If B_1 is allocated with C_3 , and B_2 with C_4 , the outcome is a 1-resilient computation.

2.3 Conditions and Formulation

Let each executable object instance p have a set of resource requirements $\{R_i^{(p)}\}$ and service requirements $\{S_i^{(p)}\}$, called its *dependency set*, which we denote as DS_p . Restricting p with a time constraint TC_p implies a projected time constraint to each member of its dependency set. Each projection is a result of the temporal relation between p 's execution and its requirements. A service requirement can be executed by another executable object instance, which can be chosen out of a set of alternatives. Hence, we can define the dependency set as follows.

Definition 1 The dependency set of an object p with a time constraint TC_p is

$$DS_{p,TC_p} = \{ \{ < R_i^{(p)}, TC_{R_i^{(p)}} > : 1 \leq i \leq k \}, \{ S_i^{(p)} : 1 \leq i \leq n \} \}.$$

where

$$S_i^{(p)} = \{ < s_j^{(p)(i)}, TC^{(p)(i)} > : 1 \leq j \leq M^{(p)(i)} \}$$

and $M^{(p)(i)}$ is the number of service alternatives of service requirement $S_i^{(p)}$.

Consider a graph that models the dependency relations between an object p and its requirements, denoting each relation by a directed arc from an object to a member of its dependency set. If a member of the dependency set is another object q , then q 's dependency graph is a sub-graph of p 's dependency graph.

Definition 2 The dependency graph of an object is a graph in which the object is represented as a node, and directed arcs connect this node to the dependency graphs of the members of its dependency set.

We can also define the set of members in each sub-graph as follows.

Definition 3 A reachability set of an object p_i is the set of all objects p_k , such that there exists a finite path from p_i to p_k in the dependency graph of p_i .

In [7] we have shown that for non-preemptive scheduling discipline we require a totally-disjoint (\bowtie) relation between all the computations, as well as avoiding conflicts between windows of occurrence and their corresponding computation non-convex intervals.

In the following definitions, we use notation which we adopted in [7]. Each convex interval, say A , is assumed to be delimited by t_a^A and t_b^A . The leftmost convex sub-interval of a non-convex interval B is denoted as $\triangleleft B$, and respectively the rightmost one by $\triangleright B$. \bowtie is the interval cover operator. Finally, the interval containment property is denoted by \gg , the interval intersection operator by \sqcap , and the empty set by ϕ .

Definition 4 The laxity of a (computation) non-convex interval P_i that is constrained within a (window) convex interval TC_i is defined by the pair $(x_{-}^{TC_i}, x_{+}^{TC_i})$, such that

$$x_{-}^{TC_i} = t_a^{P_i^L} - \text{begin}_{\min}(P_i^L) \leq t_a^{P_i^L} - t_a^{TC_i} = t_a^{\omega P_i} - t_a^{TC_i},$$

$$x_{+}^{TC_i} = \text{end}_{\max}(P_i^R) - t_b^{P_i^R} \leq t_b^{TC_i} - t_b^{P_i^R} = t_b^{TC_i} - t_b^{\omega P_i},$$

where $P_i^L = \triangleleft P_i$, $P_i^R = \triangleright P_i$. \square

Defining the laxity, one could phrase the condition for non-preemptive schedulability.

Condition 1 Let the incoming time constraint have an occurrence window TC_{in} and a non-convex computation requirement P_{in} . Let V be the verification interval, derived from the duration of a time constraint TC_z , for which $TC_z = TC_{in}$ or $TC_z \gg TC_{in}$, such that I° , the set of already accepted time constraints that intersect with the verification window V , satisfies

$$\exists TC_i^{(j)} \in I^\circ : TC_i^{(j)} \gg TC_z.$$

The incoming time constraint is non-preemptively schedulable if

$$\forall i \forall j : TC_i^{(j)} \in I^\circ : \exists x_{-}^{TC_i^{(j)}} \geq 0, \exists x_{+}^{TC_i^{(j)}} \geq 0,$$

$$\exists x_{-}^{TC_{in}} \geq 0, \exists x_{+}^{TC_{in}} \geq 0 : \forall P_{i_o}^{(j)} \in P^\circ : P_{in} \hat{\bowtie} P_{i_o}^{(j)}$$

where P° is the set $\{P_{i_o}^{(j)} \mid TC_i^{(j)} \in I^\circ\}$. \square

In the preemptive schedule, we now take into account the non-convex nature of the computation intervals.

Definition 5 The set of maximal convex subintervals of convex time intervals A and B is defined as $S(\{A\}, \{B\})$, such that

- $A \cap B = \emptyset \implies S = \{A, B\}$
- $A \cap B \neq \emptyset \implies S = \{A \cup B\}$.

The set of maximal convex subintervals of a non-convex time interval D is the set of maximal convex subintervals of all its convex members $\{d_i\}$. \square

Defining the set of maximal convex subintervals, one could phrase the condition for preemptive schedulability.

Condition 2 Let the incoming time constraint be a time constraint with an occurrence window TC_{in} and a non-convex computation requirement P_{in} . Let V be the verification interval, derived from the duration of a time constraint TC_z , for which $TC_z = TC_{in}$ or $TC_z \gg TC_{in}$, such that for I° , the set of already accepted time constraints that intersect with the verification window V , there exists no time constraint that contains TC_z . Let I_V be

$$I_V = I^\circ \cup \{TC_{in}\} - \{TC_z\}.$$

The incoming time constraint is preemptively schedulable if

$$\forall s_v \in S(J_V) : \sum_{v_i: TC_i \in J_V} \sum_{k: P_i^{(k)} \in P_i} \|s_v \cap P_i^{(k)}\| \leq \|s_v\|$$

$$\wedge$$

$$\sum_{v_i: s_v \in S(J_V)} \|s_v\| \leq \|V\| - \sum_{k: P_i^{(k)} \in P_i} \|P_i^{(k)}\|$$

where s_v are the convex subintervals of $S(J_V)$. \square

The schedulability conditions, Conditions 1 and 2, establish conditions for an object allocatability.

Condition 3 An object p is allocatable, if it is schedulable, its resource requirements are schedulable, and for each of its service requirements there is at least one allocatable service alternative (in case the set of its service requirements is not an empty set). \square

The above definition is recursive, implying that there must exist at least one object with an empty service requirement set in the reachability set of each allocatable object.

We now define the resilience of an allocated computation to transient faults and to monotonic faults. But in order to do so, we first define two special subgraphs of an allocated dependency graph.

Definition 6 An allocation graph of an object is a sub-graph of the object's dependency graph in which only allocatable objects and the schedulable resources are represented.

Note that when the allocation graph of object p includes the object p itself, it contains also all the resource requirements and all the service requirements of p , due to the allocatability property.

Definition 7 An allocation alternative of an object is a sub-graph of the object's allocation graph in which for every service requirement only one service alternative is represented.

Due to the definition of the allocation graph, the service alternatives represented in the allocation alternative are obviously allocatable.

Condition 4 An allocation for the execution of object p is n -resilient to monotonic faults if p is allocatable, and there exist at least $n + 1$ distinct allocation alternatives whose intersection with each other contains at most the node p . \square

It should be emphasized that this condition does not allow any resource requirement of p or any service requirement of p to be contained in the intersection.

Definition 8 An allocatable instance of an allocation alternative A_p of an object p is the tuple (A_p, TC_p) , where TC_p is the particular time constraint reserved for this allocation alternative.

Using the above definition and recalling that a physical redundancy is also a temporal redundancy, we have the following condition.

Condition 5 An allocation is n -resilient to transient faults if the computation is k -resilient to monotonic faults, and each of its $k + 1$ allocation alternatives, $0 \leq i \leq k$, have r_i distinct allocatable instances, such that

$$\sum_{i=0}^k r_i \geq n.$$

The practical implication of the above condition can be stated informally in terms of the following allocation philosophy. An allocator may be required to achieve an objective of a given resilience to transient faults, and the number of distinct allocatable instances at a given allocation alternative cannot support it. Then, allocating additional allocatable instances of another allocation alternative will be an adequate choice for this allocator.

3 Allocation Algorithm

In this section we introduce our allocation algorithm, based on the definitions and conditions we have introduced above. But before providing the detailed algorithm, we introduce the principles according to which it works. In addition, a condensed version of the algorithm is provided for a better understanding of the principles.

Considering the dependency graph defined in section 2.3 we call a *leaf node*, a node whose service requirement set in its dependency set is empty. Recall that in the dependency graph the nodes are executable objects to be allocated. As we will see later, a leaf node plays a special role in this allocation algorithm, and is in charge of generating the "yes" answers.

The state of an executable object during allocation can be *allocatable* or *non-allocatable*. An executable object is *allocatable* when it satisfies the allocatability condition as specified in Condition 3. Even when an executable object is non-allocatable, it is assumed to be capable to respond to the algorithm performed by the allocator.

We assume that the allocation algorithm is performed by allocators, each of which is invoked to test the satisfaction of Condition 3 by a particular object². Therefore, we start by defining the invocation messages used in the algorithm, and we go to describe the principles of the algorithm.

3.1 Message Types Used

- *ALLOCATE*(*from, whom, TC, physical_redundancy, temporal_redundancy, to*) is the initiator message:

from - initiating object Id.
whom - set of alternative object_SAPs to be allocated.
TC - time constraint.
physical_redundancy - degree of physical redundancy.
temporal_redundancy - degree of temporal redundancy.
to - receiving allocator Id.

²The assumption is not restricting the generality of the algorithm, but rather enriches its possible implementations. Allocators can be different instances of the same allocator (e.g. a recursive call), or different allocators executing concurrently.

- *ALLOC_REQ(of,tag,from,whom,level,TC,to)* is the query message:

of - initiator Id.
tag - tag number of this *of*'s computation session.
from - object_SAP that requests the service.
whom - object_SAP whose service is requested.
level - degree of temporal redundancy requested.
TC - time constraint.
to - receiving allocator Id.

- *ALLOC REP(color,of,tag,from,whom,Δlevel,TC,to)* is the feedback message:

color - yes/no.
of - initiator Id.
tag - tag number of this *of*'s computation session.
from - object_SAP that replies.
whom - object_SAP which requested the service from *from*.
Δlevel - degree of temporal redundancy in debt.
TC - time constraint.
to - receiver Id.

3.2 Principles of Algorithm for Initiator

The following algorithm is implemented as an interface between the user who wants to initiate an allocation session and the allocator. It can be a special service-access-point of the allocator, or a dedicated server of another type. When one initiates an allocation session, one must specify its fault tolerance objectives, its set of alternatives in which the computation can be carried out, and the timing constraints for this computation. The initiating algorithm tries to reach the fault tolerance objectives by requesting allocation of computation alternatives (from the set defined above) that adhere to the physical and temporal redundancy defined by the user, as well as to the timing constraints. Tagging the alternatives allows concurrent allocation of dependency graphs while maintaining null intersection between these graphs, as long as the computation *Id* and the graph *tag* are spread with the requests throughout the graph.

Therefore, the initiator (*me*) must send enough *ALLOC REQ(...)* messages to allocate members of the alternative set defined by "whom", and *me* now has to wait for the answers. In order to have a higher degree of concurrency, an artificial object-joint is created instead of keeping *me* active while waiting (recalling that *me* can be an allocator), to collect the answers when they arrive, and to allow choosing another alternative when the answer is negative.

Decrease of physical redundancy is implicitly prevented by the algorithm. The physical redundancy is controlled through the *INSERT_TC* function (see section C.1) that does not reserve in a particular calendar two requests with the same *Id* and different *tags*. This property adheres to the null intersection requirement in Condition 4.

- Upon receiving *ALLOCATE(from,whom,TC,physical_redundancy,temporal_redundancy,me):*

1. Create an object (ROOT) whose dependency set consists of an empty set of resource requirements, and a set of service requirements whose cardinality equals the physical redundancy level required
+ 1. Distribute the alternatives of *whom* into these service requirements.
2. For every service requirement in ROOT do:
 - Select the first service alternative in the service requirement.
 - Send ALLOC_REQ for allocating the selected service alternative, distinguishing each service requirement with a different tag. The ALLOC_REQ asks for the temporal redundancy required, imposes the requested time constraint, and designates ROOT as the initiator of the allocation request.

3.3 Principles of Algorithm for Allocator

The following algorithm is implemented in all instances of an allocator object in the system. It consists of actions responding to an *ALLOC_REQ(...,me)* message (allocation request), and actions responding to an *ALLOC REP(...,me)* message (allocation reply).

An executable object (*whom*) for which an allocator receives a *ALLOC_REQ(...,whom,...,me)* message must have the *schedulability* property for itself and for its resources for each of its "to-be-executed" instances. If it is schedulable, it forwards *ALLOC_REQ(...)* messages to allocate its service requirements in its dependency set. This *forward wave* of *ALLOC_REQ(...)* messages proceeds, propagating the *ALLOC_REQ(...)* messages, until a requesting message reaches either an executable object which is *non-schedulable* or a leaf executable object which has no service requirements.

The timing constraints sent in the *ALLOC_REQ(...)* messages to the service requirements and the timing requests imposed on the resource requirements are projections of the incoming timing constraint. These projections are done according to the required temporal relations between the invoker's constraint and those imposed on the requirements. We assume that these relations are known in advance, and that they are *convergent*, as defined below.

Definition 9 A convergent temporal relation sequence, is a sequence of temporal relations (R_{xy}, \dots, R_{yz}) that satisfies

$$xR_{xy}y \dots R_{yz}x \Leftarrow x \vee xR_{xy}y \dots R_{yz}x = x \vee xR_{xy}y \dots R_{yz}x \uparrow x \vee xR_{xy}y \dots R_{yz}x \downarrow x$$

for time intervals x, y . \square

Now we examine how the *ALLOC REP(...)* messages are generated. If an executable object *whom* is requested to be allocated, and it verifies itself or its resources to be *non-schedulable*, then there is no point in verifying the *allocatability* of its resource requirements. It generates an *ALLOC REP(no,...)* message to the object which requested its service. On the other hand, if a leaf object *whom* is requested to be allocated, and it verifies itself and its resources to be *schedulable*, having no resource requirements, it generates an *ALLOC REP(yes,...)* message to the object which requested its service.

The *backward wave* of *ALLOC REP(...)* messages propagates in the following way. If both an executable object and its resource requirements have been found *schedulable* and if this object has received *all* the answers it expected with a positive color, then it sends back a positive answer message *ALLOC REP(yes,...,prev,...)* to the object that had requested its services. Thus, each node performs a boolean AND of all the positive answers. On the other hand, if a requesting object exhausts all the alternatives for any particular

service request, then it cannot meet its requirements, and it sends back a negative answer message *ALLOC.REP*(*no*,...,*prev*,...). Since in the latter case some services might have already been reserved (in particular the object itself and the resources), these reservations must be removed to release them for other possible requests.

- Upon receiving *ALLOC.REQ*(*of,tag,from,whom,temporal_redundancy_level,TC,me*)::
 1. Iterate *my_level* successful iterations, trying to reserve an execution interval for *whom* in its calendar, and for its resource requirements at their calendars. The number of iterations is bounded by the required temporal redundancy level.
 2. If no iteration was successful, send *ALLOC.REP* answering *no*.
 3. Otherwise, if *whom* is a leaf-object (having no service requirements), send *ALLOC.REP* answering *yes*, indicating how many missing temporal redundancy instances there are according to *my_level*.
 4. Otherwise (not being a leaf-object) do the following for every service requirement in *whom* dependency set.
 - Select the first service alternative in the service requirement.
 - Send *ALLOC.REQ* for allocating the selected service alternative, asking for the temporal redundancy *my_level*, projecting the proper time constraint according to the temporal relation between *whom* and the service.
 5. Update *whom* joint to include the proper information needed to deal with replies.
- Upon receiving *ALLOC.REP*(*color,of,tag,from,whom,Δlevel,TC,me*)::
 1. If the color is *yes*, and all the required temporal redundancy instances have been allocated, then mark this service requirement *done*.
 2. Otherwise, not having enough temporal redundancy instances, if there is another possible service alternative in the service requirement, do the following.
 - Select the next service alternative in the service requirement.
 - Send *ALLOC.REQ* for allocating the selected service alternative, requiring the unsatisfied temporal redundancy level (up to *my_level*), projecting the proper time constraint according to the temporal relation between *whom* and the service.
 3. However, if there are no more service alternatives at that service requirement, the following two cases are distinguished.
 - If no alternative at all at that requirement have been allocated, then send *ALLOC.REP* answering *no* to the object that required the service of *whom*. In that case release *whom*, its resources, and the rest of the requirements.
 - If some alternatives at that requirement have been allocated, then decrease the level of temporal redundancy viewed by *whom*, to the lowest between its current view and the view seen by *from*. Then, mark this requirement as *done*.
 4. If all service requirements are *done*, send an *ALLOC.REP* to with positive answer to the object that required the service of *whom*, indicating the level of temporal redundancy as limited by *whom's* view or its requirements.

We note the way in which the degree of temporal redundancy is maintained, in order to satisfy Condition 5. The temporal redundancy achieved by the object itself and its resources is bounded by the one requested from the service requirements. If a service alternative cannot satisfy the degree required by a requestor object, an additional alternative is invoked to satisfy the debt, and so on as long as there are alternatives. The sum of the redundancy achieved by the alternatives of a service requirement establishes the degree of that service requirement. The lowest degree achieved by a member of the service requirements is the one reserved and the requestor is informed about the debt. That way the requestor can try and increase the degree by requesting another alternative. The principle here is to use a physical redundancy when no more temporal redundancy can be achieved.

3.4 Local and External Variables

In the algorithm presented here we use some of the variables defined for the joint of an object (see Appendix A) and the following local variables:

my_level : the degree of temporal redundancy of this object so far.
 $\text{my}\Delta\text{level}$: the debt in temporal redundancy of this object.
 Δlevel : the debt in temporal redundancy of the service requirement.
 LM_O_K : true as long as this object's schedulability is not contradicted.
 R_is_O_K : true as long as these resources' schedulability is not contradicted.
 TC_{me} : time constraint of this object.
 R_i : a resource requirement.
 \mathcal{R}_{R_i} : the temporal relation between this object and resource requirement R_i .
 TC_i : the time constraint of the requirement as projected from TC_{me} using the temporal relation \mathcal{R} .
 S_i : a service requirement.
 \mathcal{R}_{S_i} : the temporal relation between this object and service requirement S_i .
 $s_j^{(i)}$: a service alternative of service requirement S_i .
 (k, n) are the number of requirements for resources and services, respectively.)

3.5 The Allocation Algorithm

Upon receiving $\text{ALLOCATE}(\text{from}, \text{whom}, \text{TC}, \text{physical_deg}, \text{temporal_deg}, \text{me})$:

begin

Let whom be associated with $\{p_1, \dots, p_k\}$ with $k > \text{physical_deg}$;

Construct a non-volatile auxiliary object ROOT with the following:

1. $DS_{\text{ROOT}, \text{TC}_{\text{ROOT}}} = \{ \{R_i^{(\text{ROOT})} : 1 \leq i \leq k\}, \{S_i^{(\text{ROOT})} : 1 \leq i \leq n\} \}$
 where $\{R_i^{(\text{ROOT})} : 1 \leq i \leq k\} = \emptyset$
 and $S_i^{(\text{ROOT})} = \{ \langle s_j^{(\text{ROOT})(i)}, \text{TC} \rangle : s_j^{(\text{ROOT})(i)} \in \{p_1, \dots, p_k\} \}$.
2. $Id \leftarrow \text{ROOT}$.
3. $\text{prev} \leftarrow \text{from}$.
4. $\text{TC}_{me} \leftarrow \text{TC}$.
5. $\text{my}\Delta\text{level} \leftarrow 0$.

```
6.  $s[i] \leftarrow s_1^i$ , for  $1 \leq i \leq physical\_deg$  .  
7.  $Ans[i] \leftarrow off$ , for  $1 \leq i \leq physical\_deg$  .  
for  $tag \leftarrow 1$  to  $physical\_deg$  step 1 do  
    send ALLOC_REQ(ROOT,tag,ROOT, $s_1^{(ROOT)(tag)}$ ,temporal_deg,TC,allocator) ;  
od  
end
```

Upon receiving ALLOC_REQ(Id,tag,from,whom,temporal_deg,TC,me)::

```
begin
    my_level←0; LM_O_K←true;
    TCme ← construct(whom, TC, Id, tag);
    TCme.level ← my_level; TCme.state ← idle;
    while (my_level<temporal_deg) ∧ (LM_O_K) do
        /* Temporal redundancy reservations */
        if INSERT_TC(whom, TCme) then
            /* Reserve necessary resources for whom */
            i ← 1; R.is_O_K←true ;
            while (i ≤ k) ∧ (R.is_O_K) do
                TCi ← project (RR_i, TCme) ;
                TCi.level ← my_level ;
                if INSERT_TC(Ri, TCi) then
                    i ← i + 1 ;
                else /* cannot get them all: release guaranteed subset */
                    R.is_O_K←false ;
                    for q ← 1 to i step 1 do
                        REMOVE_TC (Rq, TCq) ;
                    od
                    I.M.O.K←false ;
                    REMOVE_TC (whom, TCme) ;
                fi
            od /* resource reservation terminated */
        else
            LM_O_K←false ;
        fi
        if (LM_O_K) then
            my_level←my_level+1 ;
            TCme.level ← my_level ;
        fi
    od
```

```

myΔlevel←my_level-temporal_deg ;
if (my_level=0) then
    send ALLOC REP(no, Id, tag, whom, from, myΔlevel, TCme, allocator) ;
else /* my_level> 0, something was reserved */
    if (∀Si ∈ DS(whom) : Si = φ) then /* leaf-object */
        send ALLOC REQ(yes, Id, tag, whom, from, myΔlevel, TCme, allocator) ;
    else /* non-leaf-object: invoke allocation of service requirements */
        prev ← from ;
        for i ← 1 to n step 1 do
            TCi ← project (RSi, TCme) ;
            send ALLOC REQ(Id, tag, whom, s1(whom)(i), my_level, TCi, allocator) ;
            s[i] ← s1(whom)(i) ; Ans[i] ← off ;
        od
        /* In case allocator is reentrant: store in whom joint */
        store Id, prev, TCme, myΔlevel, s[i = 1, ..., n], Ans[i = 1, ..., n] ;
    f   f
end

```

Upon receiving *ALLOC REP*(*color, Id, tag, from, s_j^{(whom)(i)}, Δlevel, TC, me*):

```

begin
    Restore auxiliary variables according to Id, tag, sj(i)
    /* prev, TCme, myΔlevel, s[i] = 1, ..., n, Ans[i] = 1, ..., n */
    if (color = yes) then
        if (Δlevel = 0) then
            Ans[i] ← done ;
        else /* Δlevel < 0 : more alternatives are needed, some already reserved */
            Ans[i] ← on ;
    fi   fi
    if (Δlevel < 0) ∧ (j < M(whom)(i)) then
        TCi ← project (RSi, TCme) ;
        send ALLOC REQ(Id, tag, whom, sj+1(whom)(i), -Δlevel, TCi, allocator) ;
        store s[i] ← sj+1(whom)(i) ;
    elseif (Δlevel < 0) ∧ (j = M(whom)(i)) ∧ (Ans[i] = on) then
        myΔlevel ← min(myΔlevel, Δlevel) ;
        Ans[i] ← done ;
    elseif (Δlevel < 0) ∧ (j = M(whom)(i)) ∧ (Ans[i] = off) then
        /* no alternative reserved release other requirements */
        send UNLOAD(whom, TCme) ;
        /* see section C.3 */
        if (Id ≠ whom) then /* climb up to try again */
            send ALLOC REP(no, Id, tag, whom, prev, myΔlevel, TCme, allocator) ;
        else /* ROOT failed to be allocated */
            send ALLOC REP(no, ROOT, tag, s[i], ROOT, myΔlevel, TCme, prev) ;
    fi
    else
        /* error in algorithm */
    fi
    if(∀i : Ans[i] = done) then
        if (whom ≠ Id) then
            send ALLOC REP(yes, Id, tag, whom, from, myΔlevel, TCme, allocator) ;
        else /* ROOT is properly allocated */
            send ALLOC REP(yes, ROOT, tag, ROOT, ROOT, myΔlevel, TCme, prev) ;
            /* temporal redundancy debt in myΔlevel */
            delete ROOT ;
    fi   fi
end

```

4 Properties

The major properties of the algorithm are discussed in this section: termination of the algorithm, correctness of allocatability when detected by the algorithm, the achievement of the fault tolerance objectives when allocatability is confirmed, the mutual exclusion support needed, and finally the deadlock prevention.

4.1 Algorithm Termination

We start examining the algorithm's properties by considering the possibility of a non-terminating allocation session, in cases of allocating a particular object, p , with a finite reachability set. The finite number of members in p 's dependency set, implies that there's a finite path from p to any member q in the set, and therefore within a finite time, an *ALLOC_REQ (...)* message sent from p will reach q . In addition, the finite reachability set implies a finite number of requirements. Furthermore, the finite reachability set implies that every path is either finite, or a close component, or a finite path ending with a close component. Therefore, these are the cases we examine below.

A path that starts at p , passes through one of its requirements, and is finite. It must eventually reach a leaf-node that has no requirements. There, the forward wave of *ALLOC_REQ (...)* messages is stopped, generating an *ALLOC REP (...)* message that returns on the same path used by the forward wave.

If p and its requirement q are both members of a closed component, then the following must occur. Object p inserts its incoming constraint $TC_{p(0)}$ into its calendar, reserving an interval $P_{p(0)}$ within this allowed window of occurrence. Then object p projects its incoming time constraint $TC_{p(0)}$ into a constraint $TC_{q(0)}$. Object q then inserts $P_{q(0)}$ into its calendar, and passes the request. The request continues and returns back to p , since it is a closed component. The restriction on convergent temporal relations, yields that the new arrival of the allocation request comes with a time constraint $TC_{p(1)}$ which is contained within $TC_{p(0)}$. Now p reserves another time interval $P_{p(1)}$, which is of the same duration as $P_{p(0)}$ and is definitely contained within $TC_{p(0)}$. The same argument holds for the following occurrences of forwarding the *ALLOC_REQ (...)* messages. Note that the finite interval $TC_{p(0)}$ can allow only a finite number of $P_{p(i)}$ intervals to be reserved within its limits. Once this finite number is reached, and another reservation request within this window of occurrence arrives, p is not schedulable any more (both preemptively and non-preemptively). When this case occurs, a negative reply *ALLOC REP (no, ...)* is sent back, and the forward wave is stopped.

The third case of a finite path that ends in a close component is a combination of a finite delivery of *ALLOC_REQ (...)* followed by the above scenario.

Due to the above, we can conclude that within a finite time the forward wave terminates, and only backward wave messages exist in the allocation session. Since each backward wave message uses the same path its corresponding forward wave message has passed, only in reverse direction, then we can conclude that within a finite time every *ALLOC REQ (...)* to an alternative is answered by either a positive or a negative *ALLOC REP (...)*. Having a finite number of alternatives for every requirement, and a finite number of requirements, yields the conclusion of the algorithm within a finite time.

Proposition 1 *The allocation algorithm terminates if the reachability set of ROOT is finite.* \square

In the case the graph is infinite, the algorithm also terminates due to an implicit timeout mechanism which is attached to the allocator. The dependence of allocatability on each objects and resource schedulability hides this timeout. A time constraint is not verified to be schedulable if its latest begin time has already

passed. In such a case a negative reply, *ALLOC.REP* (*no, ...*) , would be generated and forwarded to the originator. Therefore, we can state the following, expanding Proposition 1.

Proposition 2 *The search for an allocation always terminates.*

We can conclude by saying that since the algorithm always terminates, either by a normal termination or by a timeout, the answer to the originator will be generated.

4.2 Allocatability Correctness

There are only two possible "colors" for reply messages: a positive answer, the *ALLOC.REP* (*yes, ...*) message, and a negative answer, the *ALLOC.REP* (*no, ...*) message. The negative answer can be generated in two cases. The first is the case of a non-schedulable object that receives an *ALLOC.REQ* (...) message, where non-schedulability refers to the object itself or to one of its resource requirements. The second is the case of an object that receives *ALLOC.REP* (*no, ...*) answers from all its alternatives for a specific requirement, and thus is known not to satisfy the allocatability condition. The positive answer is generated in the case of a schedulable leaf node that receives an *ALLOC.REQ* (...) message and immediately answers with a *ALLOC.REP* (*yes, ...*) message. The positive answer propagates only when there were positive replies from all the service requirements of an intermediate schedulable node, and again schedulable refers to the object itself or to one of its resource requirements. Thus we can conclude that each object that sends an *ALLOC.REP* (*yes, ...*) message is either a schedulable leaf-node, and thus allocatable, or a schedulable intermediate node that received at least one *ALLOC.REP* (*yes, ...*) message from each of its service requirements, and thus is allocatable. Hence, the following proposition.

Proposition 3 *A positive reply from the allocator of ROOT, ensures the existence of a non-empty allocation graph of ROOT.* \square

4.3 Achievement of Fault Tolerance Objectives

In section 2.3, we have defined two types of redundancy, the temporal and the physical redundancy. Here we expand on these two concepts, and on their relations.

In the algorithm presented in section 3, every request for allocating an object specifies the temporal redundancy level required. The temporal redundancy level propagates with some restrictions. Assuming there is no reason to request from a service a higher temporal redundancy level than the one achieved by the requesting object, each object first attempts to reach the required level itself. If it succeeds, the request propagates with no disturbance. Otherwise only a part of the request is forwarded, whose size equals the level achieved locally (*my.level*), and a "debt" of the size

$$my\Delta level = my.level - required.level$$

is generated. Note that due to this definition *my Δ level* is non-positive. If all the service requirements achieve the redundancy level forwarded to them, then the local debt (*my Δ level*) is reported in the backward wave message. If, however, an alternative does not succeed in reaching the objectives set to it by the requestor, another alternative is invoked for increasing the temporal level. This alternative invocations continue as long as the temporal redundancy level for the service requirement does not reach the level achieved locally, and

as long as there are alternatives. If there are no more alternatives, the local level is reduced to the lowest level achieved by the requirements, and the increased debt is reported in the backward wave message.

The above procedure provides the following result: an allocatable object that answers a request *ALLOC_REQ* ($\dots, temporal_deg, \dots$) positively with an *ALLOC REP* ($yes, \dots, my\Delta level, \dots$), has reserved at least $temporal_deg + my\Delta level$ execution instances of itself and its resources, and its service requirements' answers reported on at least that amount. In other words, $temporal_deg + my\Delta level$ distinct allocatable instances are reserved. Hence the following proposition.

Proposition 4 *A positive reply from the allocator of ROOT, ensures that Condition 5 holds to satisfy a resiliency to transient faults of an allocatable ROOT of*

$$temporal_deg + my\Delta level(ROOT). \square$$

The physical redundancy is achieved by verifying that there are at least *physical_deg* allocation graphs which do not intersect each other, except in ROOT. The non-intersecting nature is achieved by maintaining that amount of distinct tags for the computation allocated *Id*. The *INSERT_TC* function that is used to verify it, assumes every object and every resource have a calendar, each of which is maintained by instances of *INSERT_TC* and *REMOVE_TC*. The implementation of the tags separation as service requirements of ROOT serves two goals. First, *physical_deg* replies with different tags are received into a boolean AND. Second, in case one alternative fails, another one can be chosen for an allocation retrial.

Proposition 5 *A positive reply from the allocator of ROOT, ensures that Condition 4 holds to satisfy a resiliency to monotonic faults of an allocatable ROOT of physical_deg. \square*

4.4 Complexity

In a wide variety of cases, real-time computations are composed of a set of objects which form a hierarchical structure. This hierarchy is depicted by allowing each object to be a member of at most one dependency set. On the other hand, each resource can be a member of more than one of the resource requirement sets in the computation session. This yields a tree-like structure for the objects which are not resources. Thus, in a graph representation of the computation, each of these objects may be connected to all the resources.

This hierarchical abstraction is widely used in real-time and object oriented systems due to the autonomy it provides. This autonomy is reached by the encapsulation of functions into the object and once triggered the object has no need for additional external stimuli. Therefore, the liveness of the invoking object is not a necessary condition to the successful completion of the invokees.

Using the above graph representation, we show now a worst case analysis of the algorithm complexity. We isolate the set of server objects *S*, such that they form a tree structure. The algorithm presented in section 3 traverses $|S| - 1$ arcs due to server dependency. Each server, in turn, may require all the resources in the resource set *R*. Therefore, the algorithm traverses $|S| \cdot |R|$ arcs in the graph due to server/resource relations. These are the only possible arcs to traverse in the above model. Hence, for hierarchical computations the complexity of the algorithm is:

$$|S| \cdot |R| + |S| - 1.$$

The time complexity is degraded if computation graphs which contradict the object architecture objectives are used.

4.5 Mutual Exclusion

The mutual exclusion of object utilization results directly from the use of calendars in each of the (totally) independent Service Access Points. For each computation session, an Id and a tag are generated in order to coordinate the different requests for computation. These request identifications are kept in the calendar of the object. From the algorithm we can see that if two or more requests come to a specific object with the same time constraints, the screening of the one that will acquire that window is done at allocation time.

All the requests use *INSERT_TC* and *REMOVE_TC* to update the calendars. These primitives ensure, in turn, that only one object has an access to the calendar of a particular requested object at a time. Therefore, at the execution time, a specific constraint is reserved to, at most, one server.

Note that the above development occurs for convex time intervals as well as for non-convex time intervals. The space in time is reserved for each convex sub-interval which is a member of the non-convex time interval. For periodic jobs, the same holds, sufficing to reserve the time constraint for each new occurrence of the periodic job at the end of an occurrence.

4.6 Deadlock

Deadlock avoidance or detection is automatic in the allocation scheme we have presented here. Note that in this section we are referring to run-time deadlocks, as opposed to allocation-time deadlocks which have been shown to be avoided in section 4.1.

Deadlock occurs when there exists a close component of waiting resources in a computation "wait-for" graph. Thus, for any deadlock scenario, the wait is for an indefinite period of time. Such a case is not possible in the allocation scheme presented here, because at run-time objects execution is carried out according to a certain non-overlapping time ordering. Therefore, objects are prevented from waiting for a resource or for a message from another object. Instead, all the resources are pre-allocated and the invocation mechanism uses an underlying message passing mechanism which involves no waits. Hence, we state the following.

Proposition 6 *There are no run-time deadlocks.*

5 Reallocation Algorithm

5.1 Rationale

Let the system resources, denoted as

$$\mathcal{P} = \{R_1, \dots, R_K\},$$

be connected with a set of physical communication links \mathcal{L}_P to form a graph

$$\mathcal{G}_P = (\mathcal{P}, \mathcal{L}_P).$$

The dependency set of every object p in the system, contains a resource requirement $\{R_i^{(p)} : 1 \leq i \leq k\}$, such that

$$\{R_i^{(p)} : 1 \leq i \leq k\} \subseteq \mathcal{P}.$$

Methods have been suggested to partition \mathcal{G}_P into clusters of resources used to monitor each other in order to detect a monotonic failure. Each of these clusters is called a *detection unit*, denoted D_i , and the

participating resources are assumed to communicate with other through a detection protocol of some kind. Here no assumption is taken about the detection protocol. However, our previous assumption on keeping calendars in a non-volatile storage, suggests a possibility of retrieving the guarantees given and not satisfied by the faulty resource.

Although we have shown that the resiliency objectives are satisfied, we suggest here an enhancement to allow recovery of the resiliency after a fault occurs. If there are unused resources in the system that can support the continuation of the execution of a physical redundancy that has failed, there is no reason for not using them. A reallocation of that alternative as a substitute to the faulty one can be easily implemented with the tools described above for the allocation. A retrieval of the calendar of the faulty resource (or object), allows invoking the reallocation with a negative ALLOC REP, and thus triggering the search of another alternative. If such an alternative is found, ROOT (and thus the owner who requested the computation) is only informed about the recovery via a positive ALLOC REP message. Otherwise, ROOT is informed with a Δ level that results from the fault.

5.2 Algorithm for Detection Unit D_i

```
Upon detecting failure (obj.sap:object, TCin:time_constraint) ::  
begin  
    inform members of  $D_i$  ;  
    retrieve calendar(obj.sap) ;  
     $\forall TC_i \in calendar(obj.sap)$  : do  
        Get auxiliary variables according to obj.sap joint ;  
        /*  $Id_i, tag_i, prev_i$  are restored */  
        my $\Delta$ level  $\leftarrow -1$  ;  
        send ALLOC REP(no,  $Id_i, tag_i, prev_i, obj.sap, my\Delta level, TC_i, allocator$ ) ;  
    od  
end
```

6 Concluding Remarks

In real-time systems, the resource management plan, the allocation, must be closely related to the scheduling, and the two are based on time considerations, rather than on a static priority scheme. The allocation presented here is fault tolerance motivated, to cope with the applications reliability goals, ensuring a user specified resiliency to failures while supporting both temporal redundancy and physical redundancy requirements. This approach allows dealing with monotonic faults and with transient faults in distinguished manner.

The allocation scheme we propose here accomplishes the hard real-time goal of guaranteeing a deadline satisfaction in case the job is accepted. In addition, this allocation scheme supports fault tolerance objectives in both damage containment and resiliency requirements. It does it in cooperation with a schedulability verification mechanism, and with an objects architecture, in which for each object there exists a calendar management that relates time to its execution. A nice feature of this scheme is the way in which it can be used for reallocation while increasing the resiliency back after a failure occurred.

The model employed in this paper has considered service requirements of an object as a boolean AND

of a boolean OR of alternatives. It has been done in an alternative selection for each requirement, while guaranteeing that all requirements are served before committing. However, other approaches can be employed with some changes in the algorithm, to support different relations between alternatives according to the application. Extending the relations may couple alternatives or exclude alternatives, according to an alternative chosen at another service. Another possible approach can be the use of OR of ANDs instead of the proposed AND of ORs. Various approaches are planned to be examined in a project of a hard real-time operating system MARUTI that is being implemented at the Computer Science Department, University of Maryland.

A Time Constraints and Auxiliary Variables in Object's Joint

```

type time_constraint = construct
    {   Id: computation identifier ;
        tag: thread indicator of computation Id ;
        level: redundancy index ;
        tc: convex_time_interval ;
        back_slack, for_slack : real ;
        P : non_convex_time_interval ;
        freq : real ;
        state : integer } ;

type resource_requirement = construct
    {    $\lambda_R$ : temporal relation ;
         $\uparrow$ resource ;
        R : non_convex_time_interval } ;

type service_alternative = construct
    {    $\lambda_s$ : temporal relation ;
         $\uparrow$ object_SAP ;
        s : non_convex_time_interval } ;

type service_requirement = set of service_alternatives ;
type schedule_type = (preemptive, non-preemptive) ;
type Answer_Wait_Indicator = (off, on, done) ;

var(obj.sap)

    calendar : ordered set of time_constraints ;
    Sched_Type : schedule_type ;
    dependency_set : set of k resource_requirements and n service_requirements ;

aux_var(obj.sap)

    wait_set: set of  $\uparrow$ object_SAP ;
     $\forall$  prev  $\in$  wait_set:
        Id : computation identifier ;
        tag : thread indicator of computation Id ;
         $TC_{ms}$  : time_constraint ;
        my $\Delta$ level : redundancy index ;
        s[n] : set of n service_requirements ;
        Ans[n] : set of n Answer_Wait_Indicators ;

```

B Detailed Review of Allocation Algorithms

B.1 Bottleneck Load Minimization Allocation Algorithm

Algorithm P-I-A (PR, IMC, AET) [2] ;

```

begin
    /* Init */
     $\overline{AET} \leftarrow \frac{1}{J} \sum_{j=1}^J T_j$ ; /* Av AET */
     $\overline{PL} \leftarrow \frac{1}{S} \sum_{j=1}^J T_j$ ; /* Av Processor load */
     $\gamma_{IMC}(i, j) \leftarrow \frac{1}{\overline{AET}} IMC_{i,j}$ ,  $1 \leq i, j \leq J$ ; /* IMC index */
     $\gamma_{PR}(i, j) \leftarrow 1 - R(\rho_{i,j})$ ,  $1 \leq i, j \leq J$ ; /* PR index */

    /* Iterate */
    for  $\alpha \leftarrow \alpha_1$  to  $\alpha_2$  step  $\Delta\alpha$  do
        for  $\beta \leftarrow \beta_1$  to  $\beta_2$  step  $\Delta\beta$  do

            /* PHASE I: combine modules with high IMC */
            /* in groups to reduce sum of processor loads */

            List  $\leftarrow$  Sort  $(p_i, p_j)$  pairs in descending IMC order ;
             $G_j \leftarrow \{p_j\}$ ,  $1 \leq j \leq J$ ;
            while List  $\neq \phi$  do
                pop  $(p_i, p_j)$  from top of List ;
                List  $\leftarrow$  List -  $\{(p_i, p_j)\}$ ;
                if  $\alpha \times \gamma_{IMC}(i, j) + \gamma_{PR}(i, j) > 0$  then
                    search  $(s : p_i \in G_s, t : p_j \in G_t)$ ;
                    if  $(s \neq t) \wedge (T_s + T_t \leq \overline{PL} \times \beta)$  then /*combine */
                         $G_s \leftarrow G_s \cup G_t$ ;  $G_t \leftarrow \phi$ ;
                         $T_s \leftarrow T_s + T_t$ ;  $T_t \leftarrow 0$ ;
                fi;
                fi;
            od;

            /* PHASE II: assign module groups to processors */
            /* and exhaustively search for smallest BOTTLENECK */

             $X \leftarrow \{G_j : 1 \leq j \leq J\}$ ;
             $L\{X\} \leftarrow \min_X \{\max_{1 \leq r \leq S} \{AET(P_r; X) + IMC(P_r; X)\}\}$ ;
            record  $L\{X\}$ ;
        od;
    od;
end;

```

B.2 Next-Fit-M Allocation Algorithm

The algorithm in [4] uses the following variables:

- T_i : task, $1 \leq i \leq n$.
- u_i : utilization factor (duty cycle) of T_i .
- $P_{i,j}$: set of tasks assigned to a processor.
- N_k : number of class- k processors used so far.

Algorithm Next-Fit-M ;

```

begin
    for  $k := 1$  to  $M$  step 1 do
         $N_k := 1$  ;
    od ;
    for  $i := 1$  to  $n$  step 1 do
         $k := \text{classify}(T_i)$  ;
            /* returns  $k$  for  $2^{\frac{i}{M+1}} - 1 < u_i \leq 2^{\frac{1}{k}} - 1$ , for  $1 \leq k < M$  */
            /* returns  $M$  for  $0 \leq u_i \leq 2^{\frac{1}{M}} - 1$ . */
        if ( $1 \leq k < M$ ) then
             $P_{k,N_k} := P_{k,N_k} \cup \{T_i\}$  ;
            if  $|P_{k,N_k}| = k$  then
                 $N_k := N_k + 1$  ;
            fi ;
        else /* ( $k = M$ ) */
            if  $\sum_{j \in P_{M,N_M}} u_j > (\ln 2 - U_i)$  then
                 $N_M := N_M + 1$  ;
            fi ;
             $P_{M,N_M} := P_{M,N_M} \cup \{T_i\}$  ;
            fi ;
        od ;
        for  $k := 1$  to  $M$  step 1 do
            if  $P_{k,N_k} = \emptyset$  then
                 $N_k := N_k - 1$  ;
            fi ;
        od ;
end ;

```

B.3 Algorithm for Relocation upon Failure Detection

In the algorithms presented in [6,5], the optimality constraint imposed on each cluster of processors is

$$m_i \leq c_i \leq M_i$$

where

$$m_i = C_i \cdot (\lambda - \epsilon), \quad M_i = C_i \cdot (\lambda + \epsilon).$$

The workload bounds on a detection unit are derived accordingly from

$$M(D_i) = \sum_{\forall n: P_n \in D_i} M_n, \quad m(D_i) = \sum_{\forall n: P_n \in D_i} m_n.$$

Each Leader maintains the following items in order to answer questions of other Leaders that cannot relocate in their own detection unit.

- $M(D_i)$,
- $m(D_i)$,
- N_i .

In addition, each Leader maintains the following items both for relocating locally (within the detection unit) and for relocating externally (moving processes to another detection unit).

- $D_i = \{P_1, \dots, P_m\}$,
- R_i the root of the subtree of τ_P corresponding to processors in D_i ,
- r_i the root of the subtree of τ_p corresponding to processes in D_i .

B.3.1 Relocation within The Detection Unit

```

Algorithm Relocate within  $D_i$  on  $P_j$  Fault ;
begin      /*  $m(D_i) - m_j \leq N_i \leq M(D_i) - M_j$  */
    do
        1. Update  $R_i$  to reflect  $P_j$  fault:
            1) delete node  $P_j$  from tree  $R_i$  ;
            2) update parent-nodes' capacities in processor cluster tree ;
        2. Update optimality bounds:
            1)  $M(D_i) := M(D_i) - M_j$ ;
            2)  $m(D_i) := m(D_i) - m_j$ ;
        3. Invoke ALLOCATE( $r_i, R_i$ ) ;
    od ;
end ;

```

B.3.2 Algorithm for Relocation to Another Detection Unit

```

Algorithm Relocate Externally to  $D_i$  on  $P_j$  Fault ;
begin
    /*  $N_i > M(D_i) - M_j$  */
    LEADER( $D_i$ ) do
        1. Notify  $\forall P_n \in D_i$  on  $P_j$  fault:
        2. Update  $R_i$  to reflect  $P_j$  fault:
            1) delete node  $P_j$  from tree  $R_j$  ;
            2) update parent-nodes' capacities in processor cluster tree ;
        3. Update optimality bounds:
            1)  $M(D_i) := M(D_i) - M_j$ ;
            2)  $m(D_i) := m(D_i) - m_j$ ;
        4. Collect network status:
            1)  $\forall n \neq i, n \in 1 \dots k$  Send status request to Leader( $D_n$ ) ;
            2)  $\forall n \neq i, n \in 1 \dots k$  collect answers (  $n, N_n, M(D_n), m(D_n)$  ) ;
        5. Ensure global balancing constraint:
             $\sum_{i=1}^k N_i < \sum_{i=1}^k M(D_i)$  ;
        6. Generate candidate set  $C$  ;
        7. Rank( $C$ ) ;
        8. Select highest ranked  $(v^*, k^*)$ , and migrate  $v^*$  to  $D_{k^*}$  ;
        9. Reflect migration and new relocation:
            1)  $N_i := N_i - W(v^*)$  ;
            2)  $N_{k^*} := N_{k^*} + W(v^*)$  ;
            3) Delete  $v^*$  from  $r_i$  and update ancestors' capacities ;
            4) if  $N_i > M(D_i)$  then goto step 6.
                /* iterate until done */
        10. Verify that the relocation holds:
            Invoke ALLOCATE( $r_i, R_i$ ) ;
    od ;
    LEADER( $D_{k \neq i}$ ) do
        1. Fetch the already allocated  $S$  and append the incoming processes:
             $S := S \oplus \{p_1, \dots, p_s\}$  ;
        2.  $r_{k^*} := \text{cluster}(S)$  ;
        3. ALLOCATE( $r_{k^*}, R_{k^*}$ ) ;
        4.  $N_{k^*} := N_{k^*} + s$  ;
    od ;
end ;

```

B.4 Heuristic Scheduling Algorithm

The following algorithm is used in the Spring Operating System, [16].

Procedure scheduler (**task_set**: task_set_type; **var** schedule: schedule_type; **var** schedulable: boolean);

```

/* task_set is the set of tasks to be scheduled. */

var EATs, EATe: vector_type;

/* Resources earliest available times in share and exclusive modes. */

begin
    schedule  $\leftarrow \emptyset$  ;
    schedulable  $\leftarrow$  true ;
    EATs  $\leftarrow$  EATe  $\leftarrow$  0 ;
    while ((task_set  $\neq \emptyset$ )  $\wedge$  schedulable) bf do begin
         $\forall T \in$  task_set: calculate  $T_{est}$  ;
        if  $\neg$ strongly-feasible(task_set, schedule) then
            schedulable  $\leftarrow$  false ;
        else begin
             $\forall T \in$  task_set: apply function  $H$  ;
             $T' \leftarrow T : \min_{T \in \text{task\_set}} (H(T))$  ;
             $T'_{est} \leftarrow T'_{est}$  ;
            task_set = task_set - {  $T'$  } ;
            schedule  $\leftarrow$  append(schedule,  $T'$ ) ;
            calculate new values for EATs and EATe ;
        end
    end
end;

```

C Scheduling Functions Used by The Fault-Tolerant Allocation

C.1 Inserting Time Constraint into Object's Calendar

```
type schedule_type = (preemptive, non-preemptive) ;

boolean function INSERT_TC (obj.sap:↑object,
    TCin:time_constraint) ;

varAlready_Accepted : set of time_constraints ;
    Sched_Type : schedule_type ;
    constraint : time_constraint ;

begin
    lock calendar(obj.sap) ;
    /* Already_Accepted ← calendar(obj.sap) */
    /* Sched_Type ← obj.sap scheduler type */
    if (∃ constraint ∈ Already_Accepted:
        TCin.Id = constraint.Id ∧
        TCin.tag ≠ constraint.tag)
    then /* prevent computation connectivity reduction */
        result ← false ;
    else
        result ← scheduler.PUSH_TC(TCin,Sched_Type) ;
        /* see [8] */
        if (result) ∧ (TCin.level ≠ 1)
            then ∀ constraint ∈ Already_Accepted | constraint.Id=TCin.Id ∧ constraint.tag=TCin.tag :
                setup consistency control (see [3,12]) to obey
                    1. identical non-determinism resolution, and
                    2. identical order of servicing input requests.
    fi
    fi
    unlock calendar(obj.sap) ;
    return(result) ;
end
```

C.2 Removing Time Constraint from Object's Calendar

```
type time_constraint = construct
    { Id: computation identifier ;
      tag: thread indicator of computation Id ;
      level: redundancy index ;
      tc: convex_time_interval ;
      back_slack, for_slack : real ;
      P : non_convex_time_interval ;
      freq : real ;
      state : integer } ;

boolean function scheduler.REMOVE_TC (obj.sap:↑object, TCin:time_constraint) ;

var Already_Accepted : set of time.constraints ;
    constraint : time_constraint ;

begin
    lock calendar(obj.sap) ;
    /* Already_Accepted ← calendar(obj.sap) */
    if (∃ constraint ∈ Already_Accepted: TCin = constraint )
        then
            ∀ constraint ∈ Already_Accepted | constraint.Id=TCin.Id ∧ constraint.tag=TCin.tag :
                rearrange consistency control (see section C.1 and [3,12]) ;
            Already_Accepted ← Already_Accepted -{TCin} ;
            result ← true ;
        else
            result ← false ;
    fi
    unlock calendar(obj.sap) ;
    return(result) ;
end
```

C.3 Loading and Unloading Time Constraint in Object's Calendar

```

type time_constraint = construct
    {
        Id: computation identifier ;
        tag: thread indicator of computation Id ;
        level: redundancy index ;
        tc: convex_time_interval ;
        back_slack, for_slack : real ;
        P : non_convex_time_interval ;
        freq : real ;
        state : integer } ;

type resource_requirement = construct
    {
         $\mathcal{R}_R$ : temporal relation ;
         $\uparrow$ resource ;
        R : non_convex_time_interval } ;

type service_alternative = construct
    {
         $\mathcal{R}_s$ : temporal relation ;
         $\uparrow$ object_SAP ;
        s : non_convex_time_interval } ;

type service_requirement = set of service_alternatives ;
type schedule_type = (preemptive, non-preemptive) ;
type Answer_Wait_Indicator = (off, on, done) ;

var(obj.sap)
    calendar : ordered set of time_constraints ;
    Sched_Type : schedule_type ;
    dependency_set : set of k resource_requirements and n service_requirements ;
aux_var(obj.sap)
    wait_set: set of  $\uparrow$ object_SAP ;
     $\forall$  prev  $\in$  wait_set:
        Id : computation identifier ;
        tag : thread indicator of computation Id ;
         $TC_{me}$  : time_constraint ;
        my $\Delta$ level : redundancy index ;
        s[n] : set of n service_requirements ;
        Ans[n] : set of n Answer_Wait_Indicators ;

local var
    Already_Accepted : set of time_constraints ;
    constraint : time_constraint ;

```

Upon receiving LOAD (obj.sap:object, TC_{in}:time_constraint) ::

```

begin
    using joint(obj.sap):
        Let DSobj.sap,TC = { {< Ri, TCRi > : 1 ≤ i ≤ k} {Si : 1 ≤ i ≤ n} }
        where Si = {< sj(i), TCSi > : 1 ≤ j ≤ M(i)} .
        for r ← 1 to k step 1 do
            TCRr ← project(Rr, TCin) ;
            CHANGE_TC_STATE ( (Rr, TCRr), active ) ;
        od
        CHANGE_TC_STATE ( (obj.sap, TCin), active ) ;
        for r ← 1 to n step 1 do
            TCSr ← project(Rr, TCin) ;
            for q ← 1 to (J: J ≤ M(r) ∧ s[r] = sj(r)) step 1 do
                LOAD( sq(r), TCSr ) ;
            od
        od
    end

```

Upon receiving UNLOAD (obj.sap:object, TC_{in}:time_constraint) ::

```

begin
    using joint(obj.sap):
        Let DSobj.sap,TC = { {< Ri, TCRi > : 1 ≤ i ≤ k} {Si : 1 ≤ i ≤ n} }
        where Si = {< sj(i), TCSi > : 1 ≤ j ≤ M(i)} .
        for r ← 1 to k step 1 do
            TCRr ← project(Rr, TCin) ;
            REMOVE_TC (Rr, TCRr ) ;
        od
        REMOVE_TC (obj.sap, TCin) ;
        for r ← 1 to n step 1 do
            TCSr ← project(Rr, TCin) ;
            for q ← 1 to (J: J ≤ M(r) ∧ s[r] = sj(r)) step 1 do
                UNLOAD( sq(r), TCSr ) ;
            od
        od
    end

```

References

- [1] Agrawala A. K. and Levi S.-T., *Objects Architecture for Real-Time, Distributed, Fault Tolerant Operating Systems*, IEEE Workshop on Real-Time Operating Systems, Cambridge MA, July 1987.
- [2] Chu W. W. and Lan L. M-T., *Task Allocation and Precedence Relations for Distributed Real-Time Systems*, IEEE Trans on Computers, Vol C-36 No 6 pp 667-679, June 1987.
- [3] Cooper E. C., *Replicated Procedure Call*, ACM Operating Systems Review, Vol 20 No 1 pp 44-55, Jan 1986.
- [4] Davari S. and Dhal S. K., *An On-Line Algorithm for Real-Time Task Allocation*, Proceedings of Real-Time Systems Symposium (IEEE), pp 194-199, December 2-4, 1986, New Orleans, LA.
- [5] Ferguson D., Kar G., Leitner G. and Nikolaou C., *Relocating Processes in Distributed Computer Systems*, IEEE Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems, pp 171-177, January 1986, Los Angeles, CA.
- [6] Kar G., Nikolaou C., and Reif J., *Assigning Processes to Processors: A Fault Tolerant Approach*, Proceedings of 14th International Conference on Fault Tolerant Computing Systems (FTCS), pp 306-309, June 1984, Kissimmee, FA.
- [7] Levi S.-T. and Agrawala A. K., *Objects Architecture: A Comprehensive Design Approach for Real-Time, Distributed, Fault-Tolerant, Reactive Operating Systems*, CS-TR-1915, Technical Report, Department of Computer Science, University of Maryland, College Park, Maryland, September 1987.
- [8] Levi S.-T. and Agrawala A. K., *Temporal Relations and Structures in Real-Time Operating Systems*, CS-TR-1954, Technical Report, Department of Computer Science, University of Maryland, College Park, Maryland, December 1987.
- [9] Liu C. L. and Layland J. W., *Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment*, Journal of the ACM, Vol 20 No 1 pp 46-61, Jan 1973.
- [10] Ma R. P., Lee E., Tsuchiya M., *Design of Task Allocation Scheme for Time Critical Applications*, Real Time Systems Symposium (IEEE), Miami Beach FA, Dec 1981.
- [11] Ma R. P., Lee E., Tsuchiya M., *A Task Allocation Model for Distributed Computing Systems*, IEEE Transactions on Computers, Vol C-31 No 1, Jan 1982.
- [12] Mancini L., *Modular Redundancy in a Message Passing System*, IEEE Trans. on Software Engineering, Vol SE-12 No 1 pp 79-86, Jan 1986.
- [13] Mok A. K. and Dertousos M. L., *Multiprocessor Scheduling in A Hard Real-Time Environment*, Proceedings of the Seventh Texas Conference on Computing Systems, pp 5.1-5.12, October 30 - November 1, 1978, Houston, Texas.
- [14] Mok A., *Fundamental Design Problems for the Hard Real Time Environment*, MIT Ph.D. Dissertation, Cambridge MA, May 1983.

- [15] Stankovic J. A. (editor), *Real-Time Computing Systems: The Next Generation*, March 1987 CMU Workshop on Fundamental Issues in Distributed Real-Time Systems, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 23, 1987.
- [16] Stankovic J. A. and Ramamrithan K., *The Design of the Spring Kernel*, Proceedings of Real-Time Systems Symposium, pp 146-157, San Jose, California, December 1987.
- [17] Zhao W., Ramamrithan K. and Stankovic J., *Scheduling Tasks with Resource Requirements in Hard Real-Time Systems*, IEEE Trans on Software Engineering, Vol SE-13 No 5 pp 564-577, May, 1987.
- [18] Zhao W., Ramamrithan K. and Stankovic J., *Preemptive Scheduling under Time and Resource Constraints*, IEEE Trans on Computers, Vol C-36 No 8 pp 949-960, August, 1987.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS N/A	
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT approved for public release; distribution unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UMIACS-TR - 88- 32 CS-TR- 2018		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION University of Maryland	6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION U.S. Army Strategic Defense Command	Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science University of Maryland College Park, MD 20742		7b. ADDRESS (City, State, and ZIP Code) Contr & Acq Mgt Ofc. CSSD-H-CRS, P.O. Box 1500 Huntsville, AL 35807-3801	800 N. Quincy Str. Arlington, Va 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DASG60-87-C-0066	N00014-87-0241
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Allocation of Real-Time Computations under Fault Tolerance Constraints.			
12. PERSONAL AUTHOR(S) Shem-Tov Levi, Daniel Mosse, and Ashok K. Agrawala			
13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) May 3, 1988	15. PAGE COUNT 45
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) <p>Allocation of resources in "next-generation" real-time operating systems requires some important features in addition to those demonstrated by current systems, resulting in an increased complexity of each system. The allocation is closely related to the scheduling, and the two are based on time considerations, rather than on a static priority scheme. The allocation is fault tolerance motivated, to cope with the application's reliability goals. Distributed system issues and adaptive behavior requirements increase the complexity and significance of the allocation approach.</p> <p>The allocation scheme we propose here accomplishes the hard real-time goal of guaranteeing a deadline satisfaction in case the job is accepted. In addition, this allocation scheme supports fault tolerance objectives in both damage containment and resiliency requirements. It does this in cooperation with a schedulability verification mechanism, and with an object architecture in which for each object there exists a calendar that maintains the time of its execution. A nice feature of this scheme is the way in which it can be used for reallocation while increasing the resiliency.</p> <p>Keywords: real-time operating systems, real-time resource management.</p>			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Ashok K. Agrawala		22b. TELEPHONE (Include Area Code) 301-454-4968	22c. OFFICE SYMBOL

